Syntactic Analysis.
Write a parser for MiniLan (III)

Theory of Automata and Discrete Mathematics
School of Computer Science
University of Oviedo

1 Evaluating expressions

The syntactic analysis is not the end of the road: once the input file, written in our programming language, is found valid according to the language specifications, a lot of things should be done. We may refer to all these remaining phases as **code generation**.

Code generation is the process by which a compiler's code generator converts some intermediate representation of source code (a parse tree) into machine code.
Of course, we are skipping all these phases. However, in order to show how these tasks can be performed, this practice will introduce the code for evaluating arithmetic expressions as it would be generated by a code generator. So, if the parser finds the expression $2 + 4 * 5 - 6 / 3 * 2$ in the input program then, after checking that the vocabulary and syntax are correct, the expression shall be evaluated and the result (`18`) shall be printed.

To do so, it is interesting to know that whenever a token is generated, **if the token has an associated value -as `NUMBER`-, the scanner might generate the token with the corresponding value matched in the input and CUP might assign the associated input value to the terminals**. Apart from this, **CUP may access and propagate this values assigning them or other generated values to the variables in the grammar**.

Thus, in the previous arithmetic expression, cup will receive `NUMBER` tokens with the following integer values attached (`2`, `4`, `5`, `6`, `3` and `2`).

Afterwards, in the parser, the production rules may refer to those values as the token's extra-information and the value obtained by evaluating the corresponding rule may be associated with the variable in the head of the rule.

For example, the subexpression `4*5` will be evaluated to `20` and this value will be associated to the variable in the root of the sub-tree (`term`).

Given that CUP generates the trees from the leaves to the root, if all the production rules that need evaluation propagate their value from the body to the head, the expressions will be also updated from bottom to top. The idea is depicted in figure 1: the values are passed from the bottom leaves to the factor nodes, from the factor nodes to the term nodes, and so forth.

So, we need to introduce **extra code** in both the lexical and syntactic specification files!

1.1 Updating the lexical specification file

If the token should have a value, then the scanner must send the token with the information. In the case of MiniLan, only the *NUMBER tokens* have an associated value, either INTEGER or REAL.

From now on, and for the sake of simplicity, both, INTEGER and REAL will be internally considered valid instances of the java *Double* class; however, any valid java class could be assigned as the associated value of the token.

The modifications to introduce for the INTEGER rule in the lexical specification files are shown in BLUE color in the following code snippet.

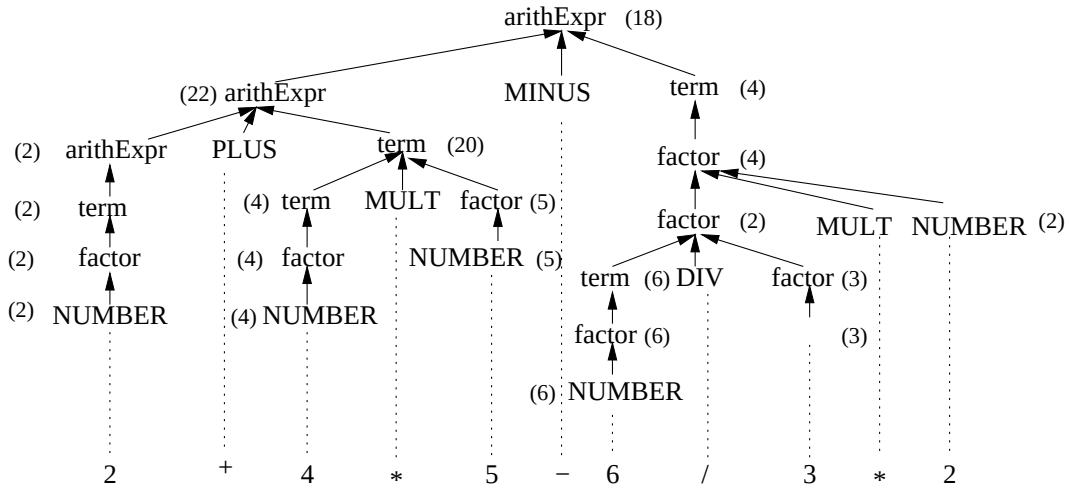

Figure 1: Evaluating a derivation tree.

```
{INTEGER}        {
                 System.out.println("SCANNER:: found NUMBER<"+yytext()+">");
                 return new Symbol(sym.NUMBER, new Double(yytext()));
                 }
```

The Symbol or token sent to the parser not only includes the token type (sym.NUMBER) but also the token value (*Double*).

1.2 Updating the syntactic specification file

This must be carefully done as there are more steps to accomplish with.

1. Assigning a data type to the nodes (either terminal or non terminal).

   ```
   terminal [<data_type>] terminal1, terminal2, ... ;
   non terminal [<data_type>] nonterminalA, nonterminalB, ... ;
   ```

   Now, the `NUMBER` token has an associated `Double` value. So,

   ```
   terminal Double NUMBER;
   ```

2. Access the nodes internal values

   To access the value associated with a variable in the body of a rule this variable must be labeled with a variable name. This name must follow Java rules to name variables and is considered a normal Java variable in every way. Once labeled, the name can be used in the associated code as any normal Java variable and refers to the value associated to the symbol (terminal or non terminal).

   ```
   factor  ::= NUMBER:v {:
          System.out.println("    PARSER::     factor <== NUMBER<"+v+">");
          :}
          ....
          ;
   ```

3. Update the value of the node.

   To update the value associated with the symbol in the head of the rule, CUP associates to each node in the parse tree an internal variable called **RESULT**, which is used to promote the values upwards in the tree. Whenever a value is assigned to `RESULT` in the Java code, this value is implicitly associated with the variable in the head of the rule.

```
factor  ::= NUMBER:v {:
       RESULT= v;
       System.out.println("    PARSER::     factor<"+RESULT+"> <== NUMBER<"+v+">");
       :}
       ....
       ;
```

Notice that, as long as `factor` will receive the value from `NUMBER` to promote it to upper nodes, we have to assign a data type to `factor`.

Thus the following modification must be done, changing the `factor` definition for including the type:

```
terminal Double NUMBER;
non terminal Double factor;
```

Some interesting comments:

- As long as NUMBER has a data type we can define a label for its associate value referring to the corresponding object.
- We can use that object later on, for instance, in the code of the production rule.
- When we assign a value to `RESULT` in the body of the rule, factor is assigned that value. So, factor has a data type, we can assign a label and so on.

2 TO DO: complete the evaluation of the arithmetic expressions

Now it is time for work. So, you must:

1. Generate the NUMBER tokens matched by the INTEGER and REAL patterns with the associate value in the lexical specification file as explained above.
2. Introduce the modifications to `factor` in the syntactic specification file. Print the values along with the previous messages.
3. Introduce the required modifications to assign values to the nodes of type `term`. Follow these steps:
   (a) assign a data type to the nodes with an associated value
   ```
   non terminal Double term;
   ```
   (b) Label nodes to refer these nodes internal values
   ```
   term ::= term:t MULT factor:f {: ...
            :}
            ....
            ;
   ```
   (c) update the value of the node
   ```
   term ::= term:t MULT factor:f {:
            RESULT= t*f;
            System.out.println("    PARSER:: term<"+RESULT+" <== term<"+t+"> * factor<"+f+">");
            :}
            ....
            ;
   ```
4. Introduce modifications to `arithExpr` in the same way.
5. Generate the parser; if no errors arise then you can validate it using this test file. If all is correct, you should notice some printed messages including factors with the right values 2, 4, 5, 6, 3 and 2!! Also, the terms and arithExprs have the corresponding calculated value with the defined operator precedences. The final arithExpr must become 18!

```
begin
  print(2 + 4*5 - 6/3*2);
end
```

6. Introduce modifications to `printSentence`, so that the result obtained from the evaluation of an arithmetic expression is printed to the screen. However, as long as the printSentence does not store any value -it just prints-, there is no need of setting a data type in the definition of this non terminal node.

   For instance, the following snipped:

   ```
   begin
     print(2 + 4*(5 - 6.0)/3*2);
   end
   ```

   should print to the screen the following text:

   ```
   PARSER:: printSentence <== PRINT (arithExpr<-0.6666666666666665>)
   ```